

Zendcon, October 19, 2009

# Writing Maintainable PHP Code

Organizing For Change

Jeff Moore

# Jeff Moore

- First programming job in 1987
- 9 years working on ERP systems
- PHP Since 2000
- Currently at Mashery (API Management)

# What is Maintainable Code?

# Maintainable Code is...

- Quick to change
- Safe to change

# Maintainability is About Change

- “The system doesn’t handle this case”
- “This is too hard to do”
- A new law goes into effect
- “Good news, we’ve been bought out”

# How Do You React To Change?

- Excitement!
- Great idea...
- We can do that...



# How Do You React To Change?

- Fear
  - That won't work because...
  - We'll do that someday...



# Much of Programming is Maintenance

- Successful small systems grow into large systems
- Systems last longer than you expect

# Organizing For Change

# How Do You Organize Your Stuff?

- Put tools and materials for a specific task in one place
- Eliminate distractions
- Have different places for different tasks

# Mise en Place

- Everything in place
- Clean up as you go
- Don't leave a mess

# Refactoring (Verb)

- Improving the design of existing software without changing its behavior
  - Adding new stuff isn't refactoring
  - Fixing bugs may not be refactoring
- Cleaning up code as we go
- Book by Martin Fowler

# Refactoring (Noun)

- A named technique for changing code
  - Split Temporary Variable
  - Extract Method
  - Inline Method
  - Replace Array with Object
  - Replace Inheritance with Delegation

# Code Smells

- Signs that you need to refactor
- Anti-Patterns
- “If it stinks, change it” - diapers or code

**Put Things that are Alike  
Together**

# Duplicated Code (Code Smell)

- When the same code exists in more than one place changes have to be made in more than one place
- Its easy to miss a change
- It takes time to analyze variation

# Shotgun Surgery (Code Smell)

- When a single change requires visiting many places in the code

# The Rule Of Threes

- First time just do it
- Second time, wince and ignore it
- Third time, eliminate the duplication

# Grouping PHP Code

- Application
- Package
- File
- Namespace
- Class
- Function
- Braces
- Whitespace

# Cohesion

# Cohesion

- How closely is the code in a group related?

# Coincidental Cohesion

- No Functions
- No Classes
- Weak relationships between code in a file

# Logical Cohesion

```
function sum_or_product($numbers, $flag) {
    if ($flag) {
        $sum = 0;
        foreach ($numbers as $num) {
            $sum += $num
        }
        return $sum;
    } else {
        $product = 0;
        foreach ($numbers as $num) {
            $product *= $num
        }
        return $product;
    }
}
```

# Signs of Logical Cohesion

- Boolean Parameters
- Switch Statements
- Unused Method Parameters

# Temporal Cohesion

- Related only by timing
- `initialize()`
- `cleanup()`
- Event handlers

# Procedural Cohesion

- Elements must be processed in order
- Otherwise, elements don't interact

# Communicational Cohesion

```
function sum_and_product($numbers) {  
    $sum = 0;  
    $product = 0;  
    foreach ($numbers as $num) {  
        $sum += $num  
        $product *= $num  
    }  
    return array($sum, $product);  
}
```

# Sequential Cohesion

```
function sum_and_average($numbers) {  
    $sum = 0;  
    foreach ($numbers as $num) {  
        $sum += $num  
    }  
    $average = $sum / count($numbers);  
    return array($sum, $average);  
}
```

# Functional Cohesion

- All lines of code work together for one purpose
- Single Responsibility Principle
- Best kind of cohesion

# Artistic Definition

“Perfection is reached, not when there is nothing left to add, but when there is nothing left to take away”

Antoine de Saint-Exupery

# Indivisible?

```
function sum_or_product($numbers, $flag) {
    if ($flag) {
        $sum = 0;
        foreach ($numbers as $num) {
            $sum += $num
        }
        return $sum;
    } else {
        $product = 0;
        foreach ($numbers as $num) {
            $product *= $num
        }
        return $product;
    }
}
```

# Replace Parameter with Explicit Methods

```
function sum($numbers) {  
    $sum = 0;  
    foreach ($numbers as $num) {  
        $sum += $num  
    }  
    return $sum;  
}
```

```
function product($numbers) {  
    $product = 0;  
    foreach ($numbers as $num) {  
        $product *= $num  
    }  
    return $product;  
}
```

# Indivisible?

```
function sum_and_product($numbers) {  
    $sum = 0;  
    $product = 0;  
    foreach ($numbers as $num) {  
        $sum += $num;  
        $product *= $num;  
    }  
    return array($sum, $product);  
}
```

# Split Loop

```
function sum_and_product($numbers) {  
    $sum = 0;  
    foreach ($numbers as $num) {  
        $sum += $num;  
    }  
    $product = 0;  
    foreach ($numbers as $num) {  
        $product *= $num;  
    }  
    return array($sum, $product);  
}
```

# Extract Method

# Rename Method

```
function sum($numbers) {  
    $sum = 0;  
    foreach ($numbers as $num) {  
        $sum += $num  
    }  
    return $sum;  
}
```

```
function product($numbers) {  
    $product = 0;  
    foreach ($numbers as $num) {  
        $product *= $num  
    }  
    return $product;  
}
```

# Indivisible?

```
function sum_and_average($numbers) {  
    $sum = 0;  
    foreach ($numbers as $num) {  
        $sum += $num  
    }  
    $average = $sum / count($numbers);  
    return array($sum, $average);  
}
```

# Extract Method

# Rename Method

```
function sum($numbers) {  
    $sum = 0;  
    foreach ($numbers as $num) {  
        $sum += $num;  
    }  
    return $sum;  
}  
  
function average($numbers) {  
    return sum($numbers) / count($numbers);  
}
```

# Small Modules

- Easy to use, call, and compose
- Help reduce duplicated code
- Encourage layered systems
- Easier to substitute implementations
- Easier to extend
- Easier to name

# Program Comprehension

Reading Code  
vs  
Writing Code

# Tools for Comprehending Code

- Brain
- IDE
- PHP Doc
- Cross Referencing Tools
- QA Tools
- Search

# Finding Definitions

- Files
- Classes
- Methods and Functions
- Properties
- Variables

# Finding Usages

- Files
- Classes
- Methods and Functions
- Properties
- Variables

# Don't Mess With Search

- `eval`
- `$$x`
- `$this->$x;`
- Concatenating variable/class/method names
- `call_user_func`

# Searching for Good Design

“Judge your architecture not by the complexity of the problems it solves, but by the ease with which you can answer simple questions about it via search.”

Jeff Moore

# Testing and Maintainability

# Testing and Maintainability

- Unmaintainable code is hard to test
- Maintainable code is easier to test
- To write maintainable code, write testable code

# What Makes Code Hard to Test?

# Cyclomatic Complexity

- The number of decision points in the code plus one
- Count each if, while, each case in a switch
- A.k.a Conditional Complexity
- Software metric for measuring maintainability?

# Conditional Complexity

```
if ($condition) {  
    $x = getA();  
} else {  
    $x = getB();  
}  
foreach ($x => $message) {  
    echo $message;  
}
```



# Long Method (Code Smell)

- Long methods are hard to maintain
- How long is too long?
- Conditional complexity  $> 10$  is too long

# Conditional Complexity and Testing

- Estimate of the number of test cases you will need to write
- The maximum number of paths through the code to achieve branch coverage

# Coverage

- Line Coverage
  - Has every line in the program been executed?
- Branch Coverage
  - Has both the true and false branch of every decision in the program been taken?

# Why Are Fewer Conditionals Easier to Maintain?

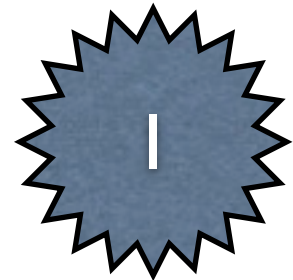
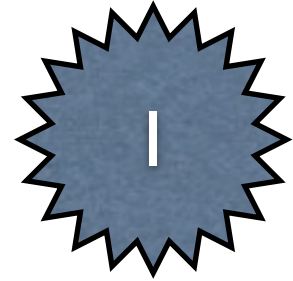
# Switch Statements (Code Smell)



```
switch ($shape) {  
  'circle':  
    $area = PI * $radius * $radius;  
    break;  
  'rectangle':  
    $area = $width * $height;  
    break;  
}
```

# With Polymorphism

- ```
class Circle {  
    function area() {  
        return PI * $this->radius ** 2;  
    }  
}
```
- ```
class Rectangle {  
    function area() {  
        return $this->width * $this->height;  
    }  
}
```



# Type Codes and Beverage Choice

```
switch ($employee_type) {  
  'programmer':  
  'manager':  
    $store->buy('coffee');  
    break;  
  'salesperson':  
    $store->buy('Lowfat soy Latte');  
    break;  
}
```

# Type Codes and Illumination

```
switch ($employee_type) {  
  'programmer':  
    $florescent_lights->off();  
    break;  
  'manager':  
  'salesperson':  
    $florescent_lights->on();  
    break;  
}
```

# Type Codes and Diversion

```
switch ($employee_type) {  
  'programmer' :  
    $browser->load("www.reddit.com");  
    break;  
  'manager' :  
    $browser->load("wsj.com");  
    break;  
  'salesperson' :  
    $browser->load("youtube.com");  
    break;  
}
```

**What code is duplicated  
in these code  
fragments?**

# Employee Class

```
class Employee {
    function arriveAtWork() {
        $this->chooseBeverage();
        $this->adjustIllumination();
        $this->procrastinate();
    }
    function chooseBeverage() {
        $store->buy('coffee');
    }
    function adjustIllumination() {
        $florescent_lights->on();
    }
}
```

# Programmer Class

```
class Programmer extends Employee {  
    function procrastinate() {  
        $browser->load("reddit.com");  
    }  
    function adjustIllumination() {  
        $florescent_lights->off();  
    }  
}
```

# Manager Class

```
class Manager extends Employee {  
    function procrastinate() {  
        $browser->load("wsj.com");  
    }  
}
```

# Salesperson Class

```
class Salesperson extends Employee {  
    function procrastinate() {  
        $browser->load("youtube.com");  
    }  
    function chooseBeverage() {  
        $store->buy('Lowfat soy Latte');  
    }  
}
```

# Testing Makes Code More Maintainable

- Maintainable code requires fewer test cases
- The path of least resistance dictates that programmers write maintainable code to avoid writing tests
- As long as you actually do write unit tests

# Testing Makes Code Safe to Change

- Having a full test suite reduces fear of change
- Tests increase confidence in the correctness of the change
- Tests increase the speed of change

**Keep Things That Are  
Different Apart**

# Unwanted Interactions

- A change in one place causes an unwanted effect in another
- Requires time researching and doing due diligence before making a change
- Risk may prevent change entirely

# Dependencies

- Defining dependencies in terms of change:
- Software element A depends upon B if “You can postulate some change to B that would require both A and B to be changed together in order to preserve overall correctness.” - Meiler Page-Jones

# Coupling

- The degree to which a software group is related to other parts of the program

# Tools for Limiting Coupling in PHP

# Variable Scopes

- Application (Global)
  - `$_GLOBALS[ 'var' ];`
- Class (Object)
  - `$this->var;`
  - `self::$var;`
- Function (Local)
  - `$var;`

# Visibility

- Private
- Protected
- Public

# Encapsulation

- Limiting interactions
- Information hiding
- Strong encapsulation groups have names
- May have a variable scope
- May specify element visibility

# What Happens When We Don't Encapsulate?

- More opportunities for interaction
- Avoid main line code
- Avoid global variables

**Keep Things That Are  
Different Apart**

# Separation of Concerns

- Put things that change for different reasons in different places
- Templating
- MVC

# Anticipating Changes

- Separation of concerns requires anticipating the likely changes
- MVC allows the view to change independently of the other layers
- MVC also requires some changes to be made in 3 places instead of 1
- Balance tradeoffs

# Untangling Dependencies

# Consider a FeedFetcher

- `Class FeedFetcher {}`
- Fetch a feed, given an Url
- This could be a performance bottle neck?

# Introducing FeedCache

```
class FeedFetcher {  
    protected $cache;  
    function __construct() {  
        $this->cache =  
            new FeedCache();  
    }  
}
```

# FeedCache Configuration

- Maximum age
- Cache directory
- Error handling policy

# How do we Configure FeedCache?

```
class FeedFetcher {  
    protected $cache;  
    function __construct() {  
        $this->cache =  
            new FeedCache();  
    }  
}
```

# Introduce Constants?

- `define('FEED_CACHE_DIR', '/tmp');`
- `define('FEED_CACHE_AGE', 3600);`
- `define('FEED_CACHE_DEBUG', true);`

# A FeedCache with Constants

```
class FeedCache {
    function __construct() {
        if (CACHE_DEBUG) {
            echo CACHE_DIR, CACHE_AGE;
        }
    }
}
```

# The Problems with Constants

- What if you need different configuration options for different feeds?
- Constants are Global
- One value for everyone

# How do we Configure FeedCache?

```
class FeedFetcher {  
    protected $cache;  
    function __construct() {  
        $this->cache =  
            new FeedCache();  
    }  
}
```

# Add Cache Configuration Options to FeedFetcher

```
class FeedFetcher {  
    protected $cache;  
    function __construct($cacheDir, $maxAge) {  
        $this->cache =  
            new FeedCache($cacheDir, $maxAge);  
    }  
}
```

# Bad API

- Bloated interface for FeedFetcher
- Can't use FeedFetcher without FeedCache

# How do we Configure FeedCache?

```
class FeedFetcher {  
    protected $cache;  
    function __construct() {  
        $this->cache =  
            new FeedCache();  
    }  
}
```

# Configuration File

- `[cache]`  
`dir = /tmp`  
`age = 3600`  
`debug = 1`

# A FeedCache with a configuration file

```
class FeedCache {
    __construct() {
        $config = Zend_Config_Ini('/path/
config.ini');
        if ($config->cache->debug) {
            echo $config->cache->dir;
            echo $config->cache->age;
        }
    }
}
```

# Punt it Up

```
class FeedFetcher {
    protected $cache;
    function __construct() {
        $config = Zend_Config_Ini('/path/config.ini');
        $this->cache =
            new FeedCache($config);
    }
}
```

# How Do We Have More Than One Kind of Cache?

```
class FeedFetcher {
    protected $cache;
    function __construct() {
        $this->cache =
            new FeedCache();
    }
}
```

# Possible Cache Classes

- File System Cache
- Database Cache
- Shared Memory Cache
- Null Cache

# The Driver Pattern

```
class FeedCache {
    static function getCache(
        $driver,
        $options = array()) {

        $file = '/drivers/' . $driver . '.php';
        require_once $file;
        $classname = ucfirst($driver) . 'Cache';
        return new $classname($options);

    }
}
```

# Defining the Driver

```
class DriverCommon {
    protected $options;
    function __construct($options) {
        $this->options = $options;
    }
}

class FileCache extends DriverCommon {}
```

# Using the Driver

```
class FeedFetcher {
    protected $cache;
    protected $options;

    function __construct($options = array()) {
        $this->cache = FeedCache::getCache(
            $options['driver'],
            $options);
    }
}
```

# Still have a problem with Configuration Options

- FeedFetcher API is still coupled to the caching configuration options
- FeedCache receives configuration options it might not care about
- Couples options together that might not be otherwise related

# Difficult to Test or Extend

- Mapping between driver name, file and class name
- Hard to inject an arbitrary driver class
- Each mock object would require a physical file

# How to Inject a Mock FeedCache for Testing?

```
class FeedFetcher {  
    protected $cache;  
    function __construct() {  
        $this->cache =  
            new FeedCache();  
    }  
}
```

# Factory Methods?

```
class FeedFetcher {
    protected $cache;

    function __construct() {
        $this->cache = $this->createCache();
    }

    function createCache() {
    }
}
```

# Use Inheritance?

```
class MyFeedFetcher extends FeedFetcher {  
  
    function createCache() {  
        return new FileFeedCache('/tmp');  
    }  
  
}
```

# Inheritance ... Not

- Heavy weight
- Wastes your one chance at single inheritance
- Cumbersome API

# Dependency Injection

# How to Configure FeedCache?

```
class FeedFetcher {  
    protected $cache;  
    function __construct() {  
        $this->cache =  
            new FeedCache();  
    }  
}
```

# Bad Options

- Configure FeedCache directly  
(go around FeedFetcher)
- Configure FeedCache through FeedFetcher  
API
- Extend FeedFetcher to configure FeedCache

# Turn the Relationship Inside Out

# Don't pull the dependency In

```
class FeedFetcher {  
    protected $cache;  
    function __construct() {  
        $this->cache =  
            new FeedCache();  
    }  
}
```

# Inject or Push the dependency In

```
class FeedFetcher {  
    protected $cache;  
    function __construct($cache) {  
        $this->cache = $cache;  
    }  
}
```

# Configuration on the Outside

```
$cache = new FileCache('/tmp');  
$fetcher = new FeedFetcher($cache);
```

# Factor Out an Interface

```
interface Cache {}
```

```
class FileCache implements Cache {}
```

```
class MemoryCache implements Cache {}
```

```
class DbCache implements Cache {}
```

# Depend on the interface, not the Class

```
class FeedFetcher {  
    protected $cache;  
    function __construct(Cache $cache) {  
        $this->cache = $cache;  
    }  
}
```

# Use Adapters to Plugin Third Party Cache

```
class CacheAdapter
    extends ThirdPartyCache
    implements Cache {
}

new FeedFetcher(new CacheAdapter(...));
```

# FeedFetcher is Now Easy To Test

- `new FeedFetcher(new NullCache());`
- No cache configuration required to test FeedFetcher
- No extension or modification to FeedFetcher required for testing

# Explicit Dependencies

- Hidden, implicit dependencies make code hard to test and hard to reuse
- Explicitly declare dependencies on the Interface of the class
- Easier to maintain

# Alternatives To Dependency Injection

# Service Locator

```
class FeedFetcher {  
    function __construct() {  
        $this->cache =  
            Locator::get( 'cache' );  
    }  
}
```

# Configuring With a Locator

```
Locator::set('cache',  
            new FileCache('/tmp'));  
  
$fetcher = new FeedFetcher();
```

# What is the Difference?

- `Locator::set('cacheDir', '/tmp');`  
...  
`$value = Locator::get('cacheDir')`
- `$_GLOBALS['cacheDir'] = '/tmp';`  
...  
`$value = $_GLOBALS['cacheDir']`
- `define('CACHE_DIR', '/tmp');`  
...  
`$value = CACHE_DIR`

# Service Locator

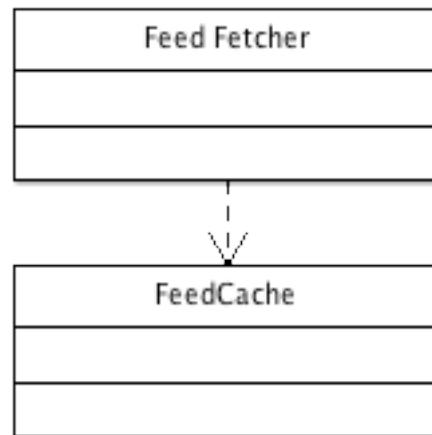
- Clients have a dependency on the locator
- Difficult to integrate locators from multiple parties
- Easier to use if you are using single instances of an object type
- Easier to use when in deeply nested code
- Less flexible

# Drawbacks to Dependency Injection

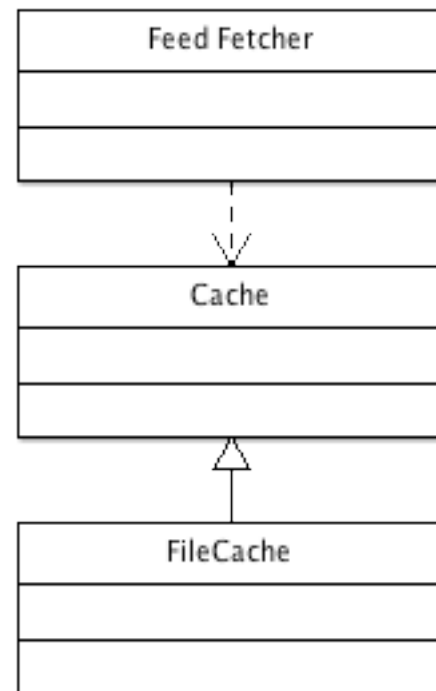
- All potential dependencies must be instantiated
- Lazy loading requires proxy objects in PHP
- Constructor methods can end up with many parameters

# Understanding Networks of Dependencies

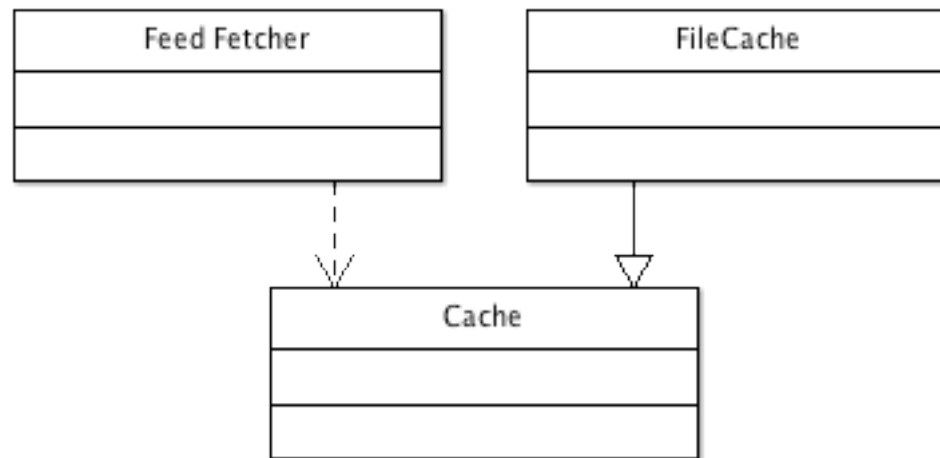
# UML Diagram



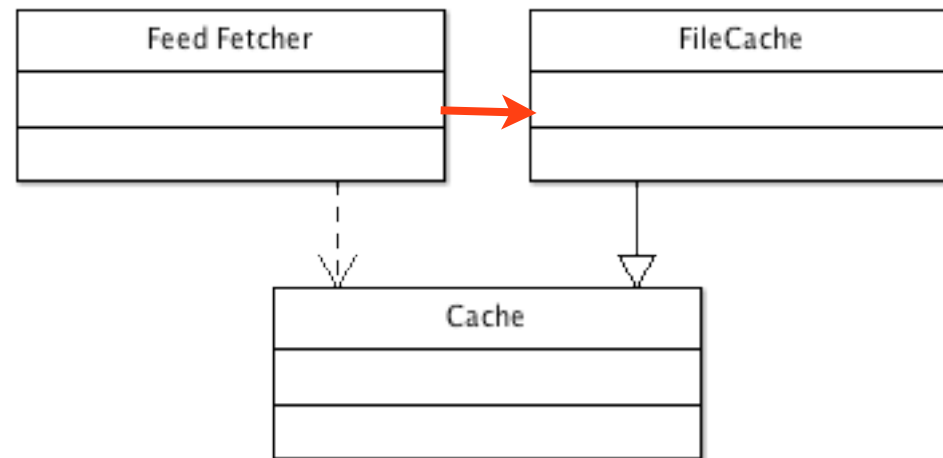
# Refactored



# Dependency Inversion Principle



# Dynamic Dependency



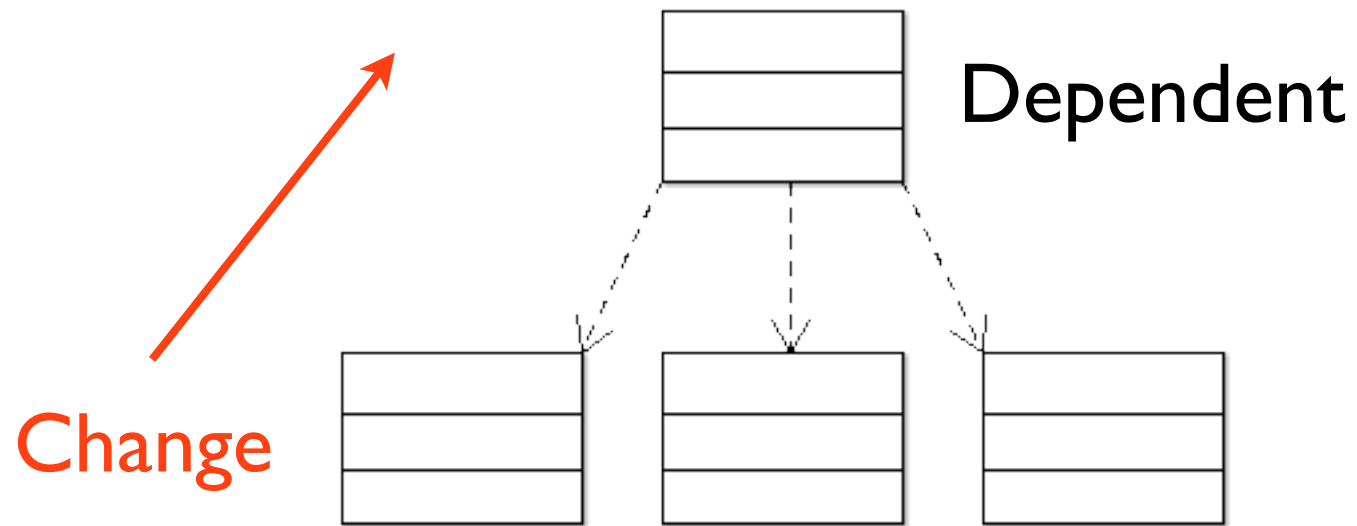
# Static vs Dynamic Dependencies

- Static arises from formal declarations
- Dynamic arises from shared assumptions
- PHP allows implied interfaces

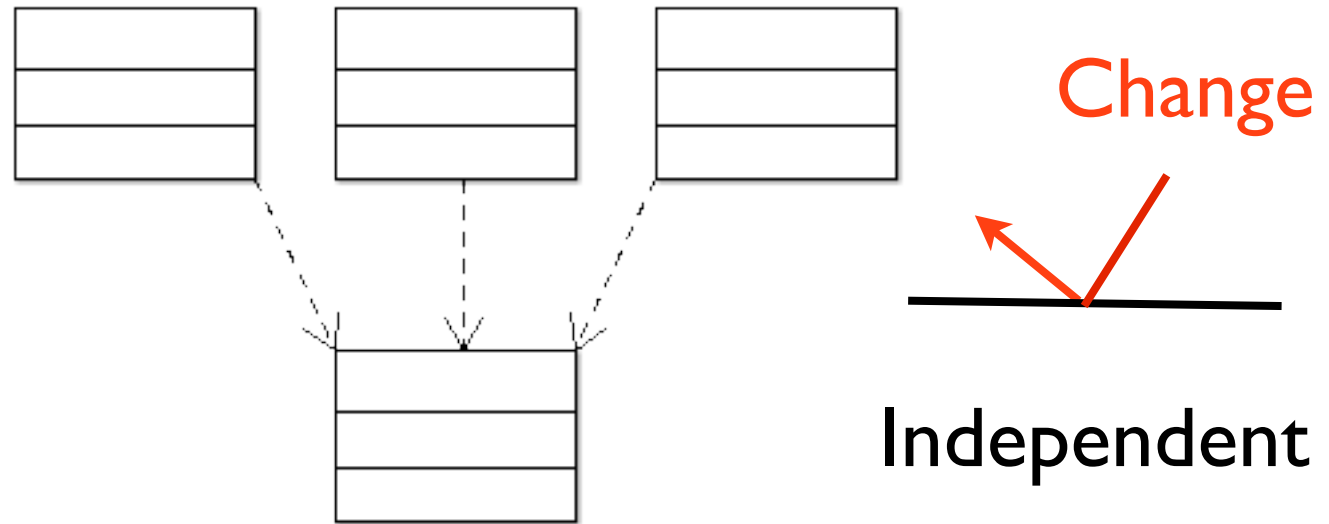
# Depending on Abstractions

- Prevents the propagation of changes across our dependency network
- Details that are unspecified are free to change
- Abstractions are little firewalls

# Dependent Class



# Independent Class

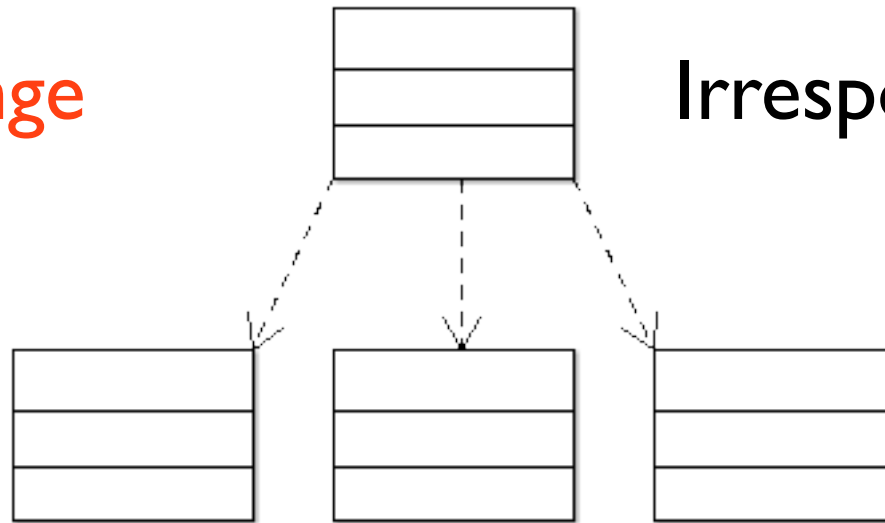


# Irresponsible Class

No where

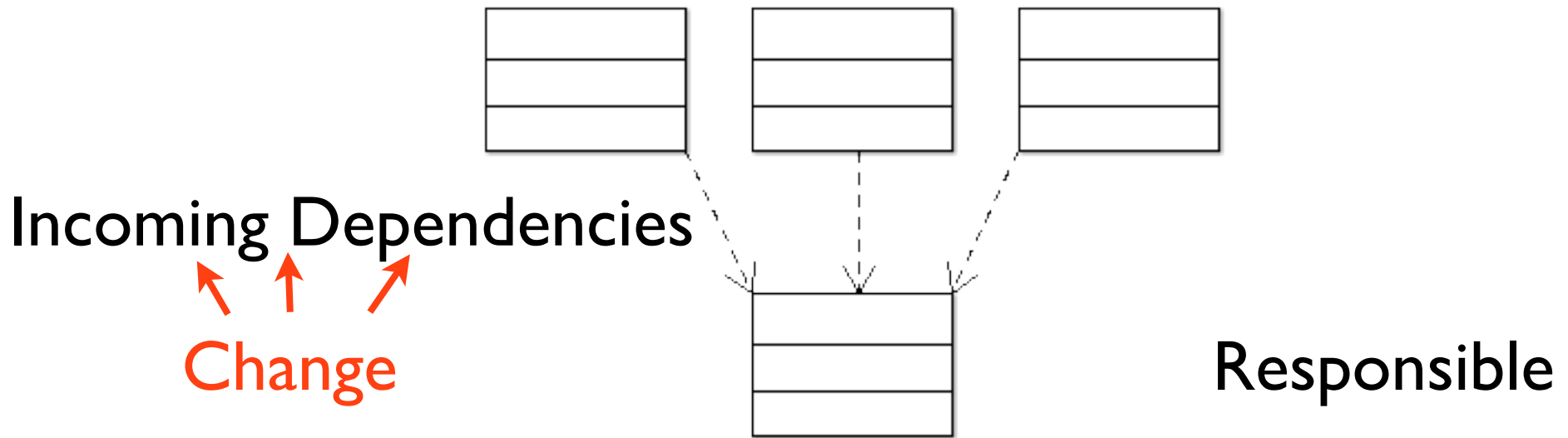
No incoming dependencies

↑  
Change

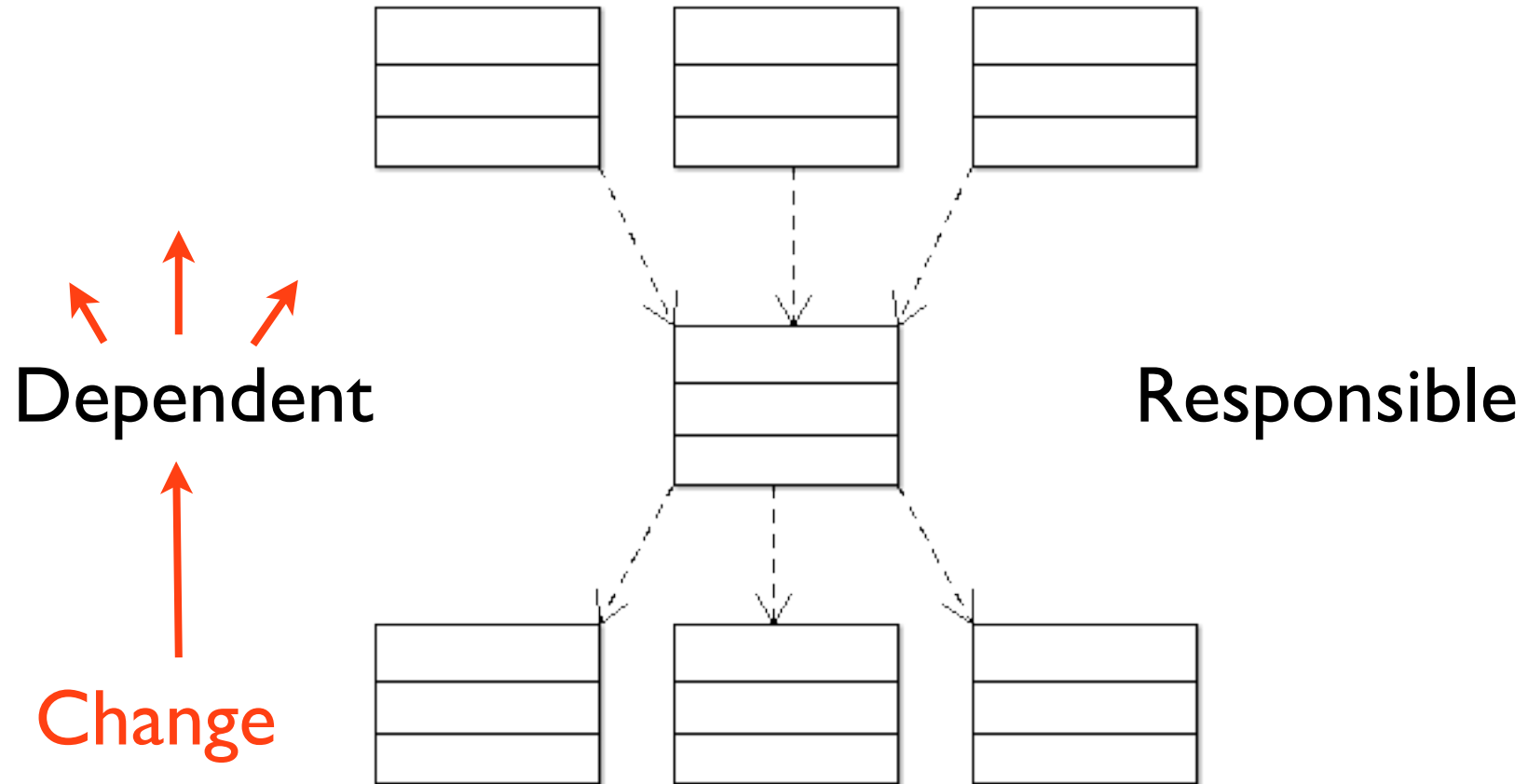


Irresponsible

# Responsible Class



# Zone of Pain



# Depend In the Direction of Stability

- Independent and Responsible
  - Design so changes don't occur here
  - Stable
- Dependent and Irresponsible
  - Design so changes occur here
  - Instable

# Anti Patterns

- Dependent and Responsible
  - Zone of Pain
- Independent and Irresponsible
  - Useless

# Too Much Abstraction

- Abstract Class
- Irresponsible
  - Few incoming dependencies
- Speculative Generality

# Favor Delegation Over Inheritance

- Refused Bequest
- `class yourclass extends ArrayObject`
- Are you really ready to support all 42 public methods in your public interface?
- The more abstract a class is, the fewer methods it should have

# Object Oriented Programming

- These concepts are not just for classes
  - Cohesion
  - Coupling
  - Encapsulation

# Which is Faster?

```
function increment($num) {  
    return $num + 1;  
}  
  
$limit = 65536*256*5;  
$counter = 0;  
  
while (($counter = increment  
($counter)) < $limit) {}
```

```
class Counter {  
    protected $counter;  
    function __construct($start=0) {  
        $this->counter = $start;  
    }  
    function increment() {  
        return ++$this->counter;  
    }  
}  
  
$limit = 65536*256*5;  
$counter = new Counter();  
  
while($counter->increment() <  
$limit) {  
}
```

# Which is Faster?

```
function increment($num) {  
    return $num + 1;  
}  
  
$limit = 65536*256*5;  
$counter = 0;  
  
while (($counter = increment  
($counter)) < $limit) {}
```



48.7

```
class Counter {  
    protected $counter;  
    function __construct($start=0) {  
        $this->counter = $start;  
    }  
    function increment() {  
        return ++$this->counter;  
    }  
}  
  
$limit = 65536*256*5;  
$counter = new Counter();  
  
while($counter->increment() <  
$limit) {  
}
```



43.6

# Good code first, performance second

- Avoid premature optimization
- Focus on value

# Refactoring

# Refactored Normalize Form

- Refactoring book becoming a standard
- Refactoring code smells away leads to consistent code
- Refactor to standard patterns
- Reveals new patterns in code that matter

# The Costs of Maintainable Code

- Maintainable code derives from revision
- Revision takes time
- Revision costs money

# When to Refactor?

# After a Change

- Fresh Familiarity
- Refactoring Fatigue
- Diminishing Returns

# Before a Change

- Refactoring Leads to Understanding
- Lack of Understanding indicates need for refactoring
- Delays Satisfaction

# When you are there

- Leave the code better than you found it

# Justifying Refactoring

- Just do it
- Do it a little bit at a time

# Writing Maintainable PHP Code

- Object oriented principles
- Refactoring
- Testing

# Questions?

- <http://www.procata.com/blog/>
- <http://www.twitter.com/selkirk/>
- <http://www.mashery.com/>
- We're Hiring

